

# Version Control (Lecture 8)

Peter Ganong, Maggie Shi, and Will Pennington

10/21/24

# Table of contents I

Conceptual

Track One Version on Local (Chapter 2.2)

Track Multiple Versions on Local – Branching (Chapter 3)

Reconcile Multiple Versions on Local – Merging (Chapter 3)

Reconciling Your Version with a Remote (Chapter 2.5 and 3.5)

Conceptual

# Roadmap

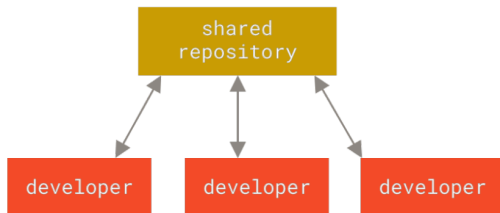
- ▶ What is version control?
- ▶ What is Git and why do I need it?
- ▶ How (not) to learn git
- ▶ Git vs. Github vs. Github Desktop

# What is version control?

- ▶ Version control: a system that records changes to a file or set of files over time so that you can recall specific versions later
- ▶ Examples of version control
  - ▶ Informal: date multiple versions of a document
  - ▶ Tools like dropbox and google docs do this automatically
  - ▶ Another neat tool similar to google docs is Google Colab
- ▶ There is now even **VSCoDe for the web**... and it uses git for version control :-)
- ▶ Cloud tools not a good version control solution for code because *code needs to run*

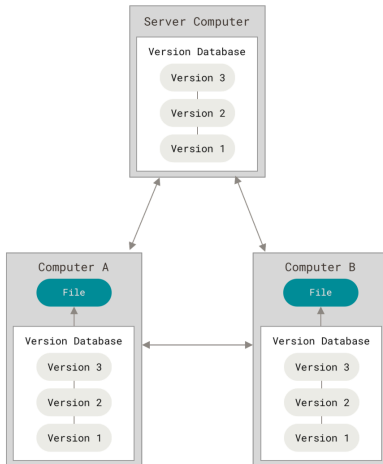
# What is centralized version control?

- ▶ Centralized version control allows for easy collaboration across developers



# Git: “distributed” version control

- Each computer fully mirrors the repository, including its full history



# Why Git?

- ▶ We will cover Git for the second half of today and next lecture. You might ask: what's in it for me?
- ▶ In general:
  - ▶ Part of the standard toolbox for analysts, but as you will see, it is not an easy skill to pick up on the job
  - ▶ Useful for solo projects for version control (as opposed to: `final_report_revised_v5_maggieedits.qmd`)
  - ▶ Crucial in group projects to make sure you don't write over others' work or break something.



# Why Git?

- ▶ In this class:
  - ▶ Use command line git to submit PS3, 4, 5, 6 and final project
  - ▶ Collaboration in PS3, 4, and final project – can work parallel as opposed to sequentially, and work without fear of writing over others' work
  - ▶ Resolve merge conflicts related to changes in the student repo

# How to learn git

Install git following the guide **here**

1. Accept that this is tricky and you will make mistakes
2. We are going to try a TON of different ways to teach this
  - ▶ graphical
  - ▶ code examples
  - ▶ do-pair-share in class
  - ▶ video game on pset
  - ▶ exercises on pset

# How not to learn git

- ▶ Command line git is what will be assessed on the quiz, the pset, and the final.
  - ▶ Worst for learning: You might previously have used Github desktop or dragged and dropped files into github.com. These tools are easy to use, but the whole process is like magic and you never really understand what's happening.
  - ▶ Also bad: Visual Studio Code's git panel is better than Github desktop and might ultimately speed you up, but don't use it right now. If you start using that then you won't understand as well what is happening under the hood.

# Textbook

- ▶ We will be following **Pro Git**, by Scott Chacon and Ben Straub.

Section	Pro Git Chapter
Conceptual	Chapter 1 (Getting Started)
Track One Version on Local	Chapter 2.2 (Git Basics)
Branching, Merging	Chapter 3.2 (Git Branching)
Reconciling with Remote	Ch 2.5 and 3.5

note: this is a 500 page textbook! Barely scratching the surface.  
Pset material goes beyond lecture, but still only scratches the surface.

- ▶ Cheatsheet **here (link)**

# Git vs. Github vs. Github Desktop

- ▶ Git: software used for version control locally (i.e., on your computer)
  - ▶ The “object” is a git repository
  - ▶ Was installed automatically when you downloaded Github Desktop
- ▶ Github: hosting service for git repositories – i.e., a place to store them online
  - ▶ There is one version of your repo online and another that is local, and you manually choose when to sync them (“push” and “pull”)
- ▶ Github Desktop: software to interact with Github
  - ▶ Introduced in 2017: “industry veterans” will not know what this is
  - ▶ Under the hood, it is doing command line git

# Summary

- ▶ Git is a distributed version control system
- ▶ The only way we know to really understand it is via the command line

## Track One Version on Local (Chapter 2.2)

# Roadmap

- ▶ Git workflow
- ▶ File lifecycle
- ▶ Basic Commands



# Git workflow

Every git repo has a hidden `.git` folder containing the database

Git does not track every file that is in the directory containing `.git`. Instead, you decide which files to track.

1. Modify files in your working tree
2. Selectively add files you want to commit to your staging area
3. Commit: permanently tracks the staged files as part of the version control system

Analogy: in the standard workflow, steps 2 and 3 happen automatically after you save. In Git, you choose when this happens to ensure you don't "save" something that breaks the code.

# File lifecycle

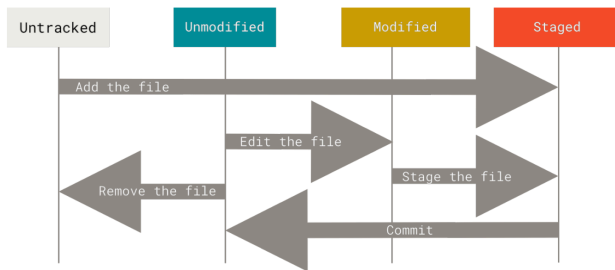


Figure 8. The lifecycle of the status of your files

# Basic Commands

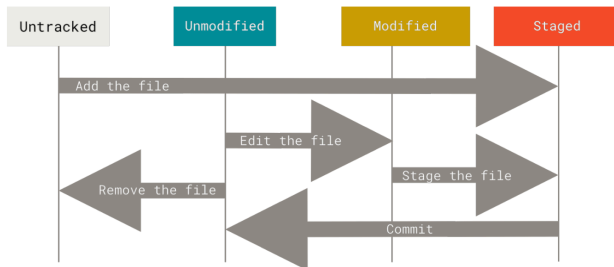


Figure 8. The lifecycle of the status of your files

\$ git status

- check status of working directory: what's staged for commit, what's tracked but not staged, and what's not tracked (but modified)

# Basic Commands

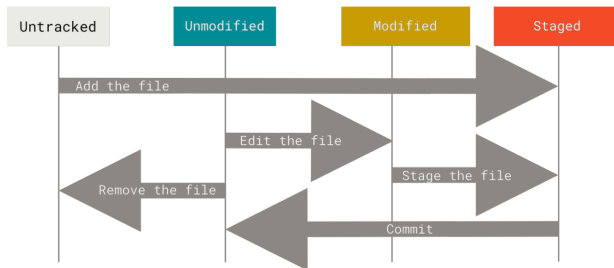


Figure 8. The lifecycle of the status of your files

```
$ git add <filename>
```

- ▶ add a single file to the staging area
- ▶ can be either the first arrow (“add the file”) or, if the file is already tracked and was since modified, the third arrow (“stage the file”)

# Basic Commands

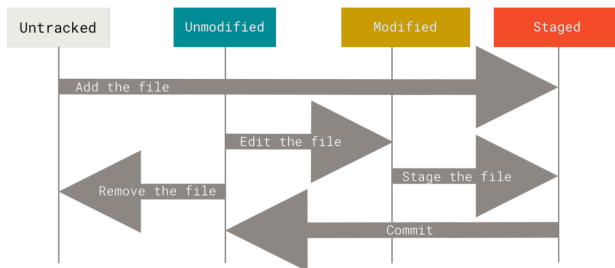


Figure 8. The lifecycle of the status of your files

```
$ git add .
```

- ▶ add all files (both untracked and modified) to the staging area

# Basic Commands

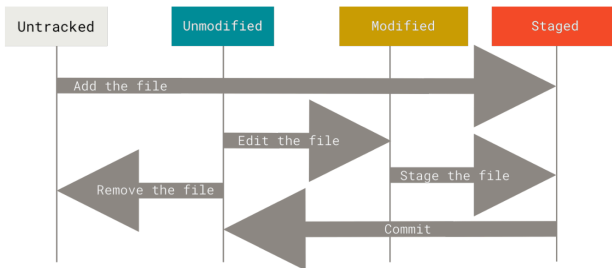


Figure 8. The lifecycle of the status of your files

```
$ git commit -m "<message>"
```

- ▶ Commit files from staging area with a message.
- ▶ The message should be informative about what's in the commit: "debugged function to do XYZ"

# Basic Commands

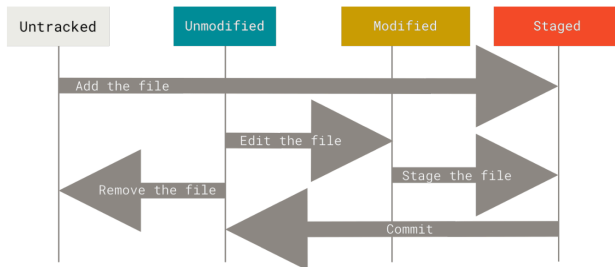


Figure 8. The lifecycle of the status of your files

```
$ git log
```

► prints log of recent commits

# Create a new repo

<https://github.com/new> and then clone to your computer

## Choices

1. Public or private?
  - ▶ Obviously anything with sensitive data or code should be private
  - ▶ Otherwise, we think it's better to leave stuff public. Public repos make it easier for a potential employer to evaluate and be confident in your coding ability.
2. License – Ganong's lab uses the MIT license. Key thing is to specify a license
3. .gitignore – next slide



# Git Ignore I

- ▶ You will often have files or filetypes that you want Git to systematically avoid (*ignore*). Examples:
  - ▶ Automatically-generated files from compiling Python (.pyc)
  - ▶ Large data files (this will become very important on PS4)
  - ▶ Mac users: .DS\_Store!
- ▶ Create a file called .gitignore to tell git which files to ignore
- ▶ You must commit your .gitignore to the repo

# Git Ignore II

Example .gitignore :

```
# ignore all .a files
*.a

# but do track lib.a, even though you're ignoring .a files above
!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO
/TODO

# ignore all files in any directory named build
build/

# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt

# ignore all .pdf files in the doc/ directory and any of its subdirectories
doc/**/*.pdf
```

- ▶ See more examples **here (link)**
- ▶ Github's **recommendation (link)** for Python. We haven't tested it yet.

# Summary

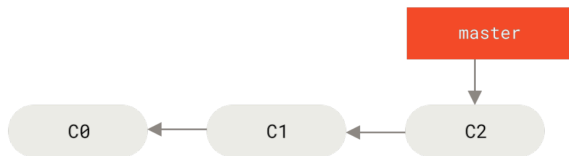
- ▶ Adding files to the staging area prepares Git to track changes
- ▶ Committing changes allows Git to take a snapshot of your files
- ▶ Use `.gitignore` to ignore irrelevant files

## Track Multiple Versions on Local – Branching (Chapter 3)

# Roadmap

- ▶ Branching: overview
- ▶ Creating a New Branch
- ▶ Switching Branches
- ▶ Other Useful Branching Commands

# Branching

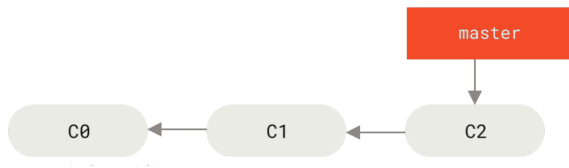


- ▶ A simple commit history

Remarks:

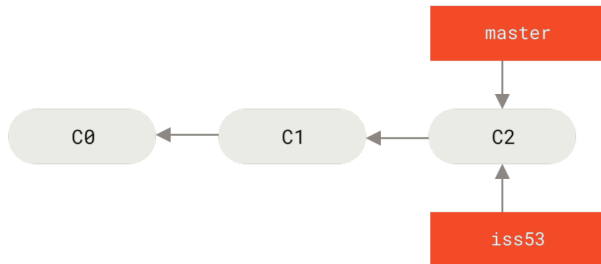
- ▶ These commit names (C0, C1...) are terrible. Insufficiently descriptive. Using them only because we are following an example from the textbook.
- ▶ Git today calls the primary branch `main`. However, git used to call the primary branch `master`. The textbook uses this legacy terminology.

# Branching



- What would you do if you wanted to do some exploratory work in a safe environment without affecting the work other team members might be doing?

## Creating a New Branch

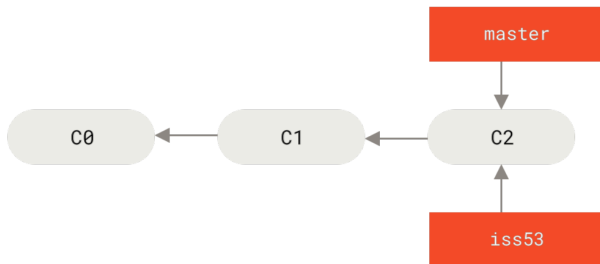


```
$ git branch iss53
```

- Creates a new branch called `iss53`



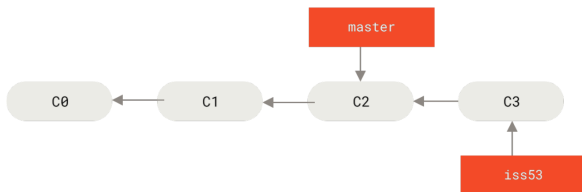
# Switching Branches I



```
$ git switch iss53
```

- Switches to your new branch `iss53`

## Switching Branches II



```
$ git commit -m "New work on iss53"
```

- ▶ With `iss53` checked out, your commits now move `iss53` forward while leaving `master` untouched

## Other Useful Branching Commands

```
$ git branch
```

- ▶ Returns a listing of your current branches

```
$ git branch -d <branch_to_delete>
```

- ▶ Deletes *branch\_to\_delete*

# Summary

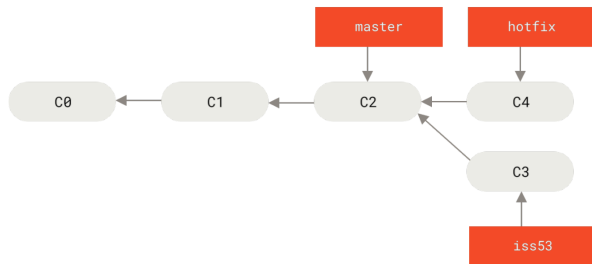
- ▶ Working in branches creates an independent, safe development environment
- ▶ Commits only modify current branch, leaving others untouched

## Reconcile Multiple Versions on Local – Merging (Chapter 3)

# Roadmap

- ▶ Fast forward merge
- ▶ Three way merge
- ▶ Three way merge with a conflict
- ▶ Do-pair-share
- ▶ Resolving file-level conflicts

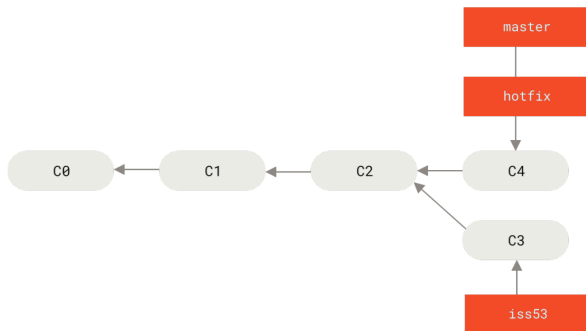
# Fast forward merge I



```
$ git switch master  
$ git branch hotfix  
$ git switch hotfix  
$ git commit -a -m "Work in hotfix branch"
```

- Suppose now that you want to switch back to master and make adjustments in a new hotfix branch

# Fast forward merge II



```
$ git switch master
```

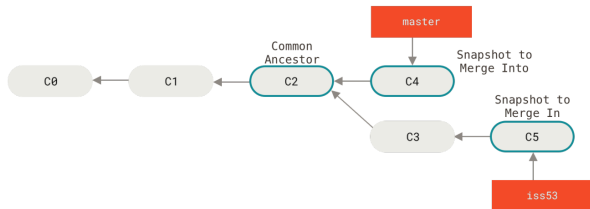
```
$ git merge hotfix
```

- ▶ Brings changes from `hotfix` into `master`
- ▶ This is called a “fast forward merge”

```
git branch -d hotfix
```

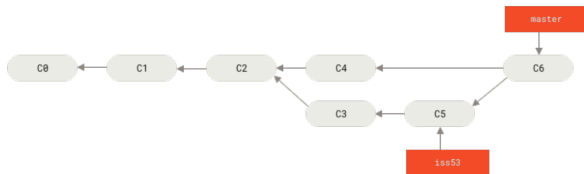


# Three way merge I



- Merging `iss53` into `master` now has to account for a divergent work history

## Three way merge II



```
$ git switch master
```

```
$ git merge iss53
```

- ▶ This is called a “three way merge” because it involves combining C2, C4, and C5 into a single unified version of the code

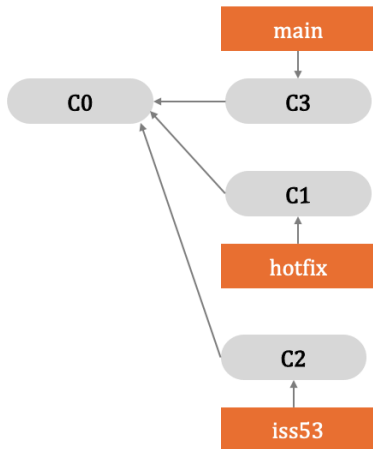
## Three way merge with a conflict I

We will use the repo **merge\_example** which has the following commit history:

name	branch	content	parent	conflict?
C0	main	Create <code>add.py</code> , which adds <code>x</code> to 3	none	no
C1	hotfix	Change <code>add.py</code> to add <code>x</code> and <code>y</code>	C0	no
C2	iss53	Change <code>add.py</code> to add a list	C1	no
C3	main	Change <code>add.py</code> to add <code>x</code> to 4	C0	no
C4	main	Merge hotfix into main	C1 & C3	yes!

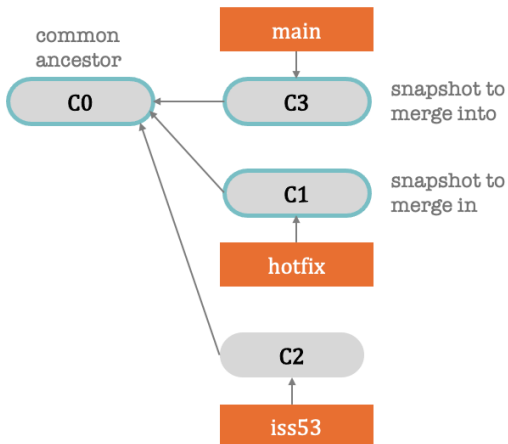
# Three way merge with a conflict II

## ► Commit history visualization



## Three way merge with a conflict III

- We will try to merge hotfix into main



## Three way merge with a conflict IV

```
'''  
A function that adds x to 4  
'''  
  
# define function  
def add(x):  
# arguments: x (number to add to 4)  
    return (x + 4)
```

\$ git switch main

```
'''  
A function that adds a list of numbers  
'''  
  
# define function  
def add(x):  
# arguments: x (list of numbers to be added)  
    return sum(x)
```

\$ git switch iss53

```
'''  
A function that adds x to y  
'''  
  
# define function  
def add(x, y):  
# arguments: x, y (numbers to add)  
    return (x + y)
```

\$ git switch hotfix

- ▶ Running *git switch* command switches to different branches, and each branch contains a different version of *add.py*

# Three way merge with a conflict V

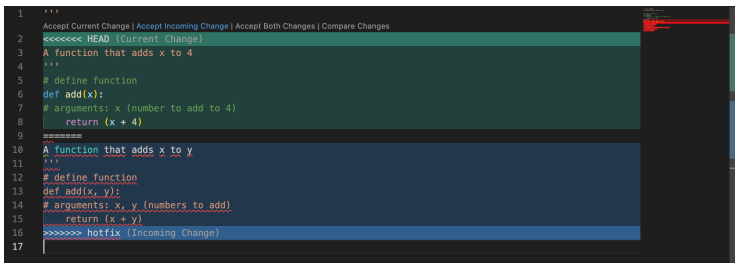
```
$ git switch main  
$ git merge hotfix
```

```
1  """  
  Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes  
2  <<<<<< HEAD (Current Change)  
3  A function that adds x to 4  
4  """  
5  # define function  
6  def add(x):  
7  # arguments: x (number to add to 4)  
8  return (x + 4)  
9  =====  
10 A function that adds x to y  
11 """  
12 # define function  
13 def add(x, y):  
14 # arguments: x, y (numbers to add)  
15 return (x + y)  
16 >>>>> hotfix (Incoming Change)  
17
```

- Merging *hotfix* into *main* causes conflicts.

# Three way merge with a conflict V

```
$ git switch main  
$ git merge hotfix
```



The screenshot shows a code editor with a dark background. The code is divided into two sections by a horizontal line. The top section is labeled 'HEAD (Current Change)' and contains a function 'add(x)' that returns 'x + 4'. The bottom section is labeled 'hotfix (Incoming Change)' and contains a function 'add(x, y)' that returns 'x + y'. The code is highlighted with green and blue colors. The editor interface includes a sidebar on the right with a file explorer and a search bar.

```
1  ...  
2  Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes  
3  <<<<<< HEAD (Current Change)  
4  A function that adds x to 4  
5  ...  
6  # define function  
7  def add(x):  
8  # arguments: x (number to add to 4)  
9  return (x + 4)  
10  
11  
12  =====  
13  A function that adds x to y  
14  ...  
15  # define function  
16  def add(x, y):  
17  # arguments: x, y (numbers to add)  
18  return (x + y)  
19  >>>>>> hotfix (Incoming Change)  
20
```

- ▶ Merging *hotfix* into *main* causes conflicts.
- ▶ Resolve within Visual Studio Code.
  - ▶ Remark: Although merge conflicts can be resolved via command line, it is more time consuming and provides no educational benefit to do so.
- ▶ Discussion question: Do we want “Current Change”, “Incoming Change” or “Both”? Why?

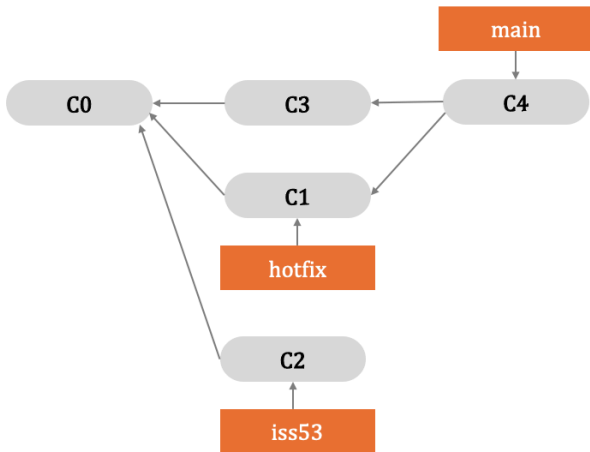


## Three way merge with a conflict VI

How add.py looks after conflict is resolved \* Resulting file in main looks exactly like version in hotfix

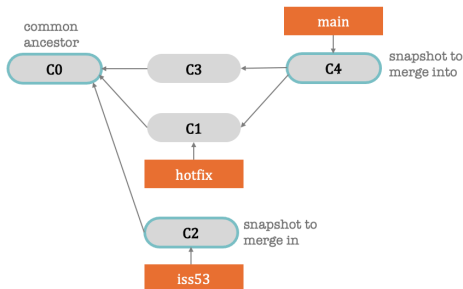
```
'''  
A function that adds x to y  
'''  
  
# define function  
def add(x, y):  
# arguments: x, y (numbers to add)  
    return (x + y)
```

## Three way merge with a conflict VII



...

## do-pair-share: merge conflicts



Goal: the file *add.py* should look identical on **main** and **iss53**

1. On github.com, fork  
[https://github.com/uchicago-harris-dap/merge\\_example](https://github.com/uchicago-harris-dap/merge_example)
2. Clone your fork to local
3. Merge **iss53** into **main**, naming your commit **C5**
4. Using VSCode, resolve the conflict in favor of the **iss53** version

# Resolving file-level conflicts I

- ▶ For some file types, line-by-line adjustments will not be possible
  - ▶ E.g. PDFs

# Resolving file-level conflicts II

## Repo

- ▶ Branch *iss1* has a version of the file *Example.pdf* that conflicts with the version on *main*
- ▶ How to resolve the conflict when merging *main* into *iss1*?
- ▶ Pull request **here**

## Resolving file-level conflicts III

```
$ git switch iss1
```

```
$ git merge main
```

▶ After merge step, git returns a conflict message.

## Resolving file-level conflicts III

```
$ git switch iss1  
$ git merge main  
$ git checkout --theirs Example.pdf  
$ git add Example.pdf  
$ git commit -m "C5: Merging main into iss1"
```

- ▶ Solution: *checkout* command with
  - ▶ `-theirs`: keep file from incoming branch (main)
  - ▶ `-ours`: keep file from current branch (iss1)
- ▶ Add changes and commit to finalize merge

# Summary

- ▶ Merging brings changes from one branch into another
  - ▶ Easier case: fast forward
  - ▶ Harder case: three way merge
- ▶ When branches have divergent commit histories, you may have to manually resolve conflicts
- ▶ Some conflicts must be resolved by keeping files from one branch or another



## Reconciling Your Version with a Remote (Chapter 2.5 and 3.5)

# Roadmap

- ▶ list all branches
- ▶ fetch
- ▶ pull
- ▶ pull requests
- ▶ push

see all branches

- ▶ list all branches, including branches in the remote repository which are not on your computer

```
$ git branch -a
```

# fetch

```
$ git fetch
```

- ▶ Get changes from remote repository to your computer.
- ▶ This command does not modify any of your existing work.
- ▶ If you did some work in the same branch, you will need to do a *git merge origin* to reconcile your work with the work in the remote.

# pull

- ▶ *git pull* equivalent to *git fetch* followed by *git merge*

```
$ git pull <branch_name>
```

- ▶ Automatically fetches branch from the remote repository, then merges that branch into your current branch
- ▶ If there are reconcilable changes in both places, git will create a reconciling merge commit and ask you to confirm the content of the git message.
- ▶ If there are irreconcilable changes, then you are in the world of the last section of lecture where you need to do some reconciliation.

# Pull Requests

- ▶ A pull request is a way to merge changes from a branch into an upstream repository.
- ▶ Doesn't `git merge` already do this? Yes, but it this way allows for collaborative peer review
- ▶ Example: open pull request for merge\_example repo (here)

Compare main and iss53 via Pull Request #1

[Open](#) whpennington wants to merge 1 commit into `main` from `iss53`

Conversation 0 Commits 1 Checks 0 Files changed 1 +3 -3

**whpennington commented now**

No description provided

C2: Change add.py to add list on iss53 7d9958

**This branch has conflicts that must be resolved**

Use the [web editor](#) or the [command line](#) to resolve conflicts.

**Conflicting files**

add.py

Merge pull request You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

**Add a comment**

Write Preview

Add your comment here...

**Reviewers**

No reviews

Still in progress? [Convert to draft](#)

**Assignees**

No one—[assign yourself](#)

**Labels**

None yet

**Projects**

None yet

**Milestones**

No milestone

**Development**

Successfully merging this pull request may close these issues.

None yet

**Notifications** Customize

[Unsubscribe](#)

You're subscribed to this repository's activity.

# Pull Requests

- ▶ Pull requests allow for line-by-line comparison of changed files
  - ▶ Click “+” to add a comment (show in browser)
  - ▶ They also identify merge conflicts

## Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#) or [learn more about diff comparisons](#).

base: main

compare: iss53

✖ Can't automatically merge. Don't worry, you can still create the pull request.

Discuss and review the changes in this comparison with others. [Learn about pull requests](#) [Create pull request](#)

1 commit 1 file changed 1 contributor

Commits on Sep 26, 2024

C2: Change add.py to add list on iss53

whperrington committed 3 weeks ago

768958 <>

Showing 1 changed file with 3 additions and 3 deletions. [Split](#) [Unified](#)

add.py

```
... 00 -1,7 +1,7 @@
1 1 ...
2 - # A function that adds 8 to 3
2 + # A function that adds a list of numbers together
3 3 ...
4 4 # define function
5 5 def sum(x):
6 - # arguments: x (number to add to 3)
7 -     return (x+3)
6 + # arguments: x (list of numbers to be added)
7 +     return sum(x)
```

# git merge vs pull requests

When should I use each?

- ▶ Use pull requests when you want to preview a change
- ▶ Use pull requests when you want someone else to review the change
- ▶ Otherwise, just use `git merge`



# push

```
$ git push <branch_name>
```

- ▶ push (committed) changes from local to remote repository
- ▶ what happens if you push and there are also changes in the remote? “Updates were rejected because the remote contains work that you do not have locally. This is usually caused by another repository pushing to the same ref. You may want to first merge the remote changes (e.g., hint: ‘git pull’) before pushing again.”

# Summary

- ▶ *fetch* brings changes from remote to local
- ▶ *pull* brings changes from remote to local and tries to reconcile via merge commit
- ▶ pull requests provide a “trial run” of merges that is visible to collaborators
- ▶ *push* sends changes from local to remote
- ▶ Remark: “sync changes” (appears in GH desktop and VSCode) really means push and pull